ESD-TR-70-141

THE DESIGN AND IMPLEMENTATION OF A CONVERSATIONAL EXTENSIBLE LANGUAGE

Jay M. Spitzen

May 1970

DIRECTORATE OF SYSTEMS DESIGN AND DEVELOPMENT
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

(Prepared under Contract No. F19628-68-C-0101 by Harvard University, Cambridge, Massachusetts.)

ESD-TR-70-141

THE DESIGN AND IMPLEMENTATION OF A CONVERSATIONAL EXTENSIBLE LANGUAGE

Jay M. Spitzen

May 1970

DIRECTORATE OF SYSTEMS DESIGN AND DEVELOPMENT
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

(Prepared under Contract No. F19628-68-C-0101 by Harvard University, Cambridge, Massachusetts.)

FORWARD

This report was prepared in support of Project 2801, Task 280102 by Harvard University, Cambridge, Massachusetts under Contract F19628-68-C-0101, monitored by John Goodenough, ESMDA, and was submitted 1 May 1970.

This technical report has been reviewed and is approved.

Sylvia R. Mayer
SYLVIA R. MAYER
Project Officer

[illegible]
for WILLIAM F. HEISLER, Col, USAF
Director of Systems Design & Development

Acknowledgment

I have found it a privilege and a pleasure to do computer science at Harvard. In particular, I wish to express my gratitude and appreciation for the guidance of two men without whom the work described here would not even have been contemplated.

Professor T. E. Cheatham introduced me to computer science. Most of what I know of the field rests on the foundation that he built, and for this I am in his debt.

Professor Thomas A. Standish has given generously of his time and effort in providing technical advice, criticizing rough drafts of this report and lending a sympathetic ear when one was needed.

There is an intangible quality in computer science - good taste - that usually makes the difference between good and bad computer science. I consider myself fortunate to have such models of this quality as Professors Cheatham and Standish to try to emulate.

Thank you.

Jay M. Spitzen

ABSTRACT

This report describes CEL, a conversational extensible language. Its syntax, data, control structures and conversational features are presented and compared to those of other languages. Its use is illustrated by means of several examples in the areas of list processing, polynomial arithmetic, formula manipulation, vector arithmetic, trees and syntax analysis, complex and rational arithmetic and block structure and own variables.

TABLE OF CONTENTS

SECTION I

Introduction

In November 1969 the author undertook the design of a conversational extensible language and the implementation of that language on Harvard University's PDP-10 Computer. That design is now complete and the language processor that was built, CEL, is now running. The purpose of this paper is to describe the language and its use.

There are today, it has been estimated, over 1700 different programming languages in over 40 special application areas. Thus he who proposes another language must either be able to show that it serves a purpose not already better served by one of its predecessors or else be judged guilty of having done no more than contributed to the flood of languages. We will argue that the language presented here is especially useful for a significant class of users and variety of uses by virtue of being both conversational and extensible. We will argue further that the properties of conversationality and extensibility are particularly suited to being combined in a single language. Thus we feel justified in unleashing CEL on the world.

SECTION II

The Limits of Conventional Languages

A. An Introduction

In computing we have theoretical constructs (e. g. the Turing Machine or Markov Algorithm) capable, if Church's Thesis is correct, of computing any result for which we can specify a sufficiently precise procedure. Yet we find that the practical problem of computing something even moderately complex is often a great strain on our talents. It may be that a reason for this distinction between what is computable in theory and what is computable in practice is the inadequacy of the tools we use to describe computation to our machines - that is, programming languages.

Programming languages range from absolute machine code and assembly languages to a wide variety of higher level languages. The former give us total flexibility and computational power and the efficiency of hand tailored models. The price we pay for this power and efficiency is that programs written in assembly languages to solve hard problems are generally of immense length and complexity. In this respect one notes that the production of the OS/360 operating system has so far taken three man millenia of effort and cost on the order of 100 million dollars. The current result is a program whose length is over 5 million lines of code and which is still not completely debugged[22].

Hence for a wide variety of problems we have sought refuge from this complexity in the higher level languages. Let us consider then the scope of applicability of various kinds of higher level languages. It is a reasonable approximation, for this purpose, to classify higher level languages as either special or general purpose and to break down the latter class into monolithic and extensible (or, to use Cheatham's terms, shell and core) languages.

B. Special Purpose Languages

Two of the best known special purpose languages are FORTRAN and ALGOL 60. These permit the production of programs that are fairly readable and short, especially by comparison with machine code, for the things it is reasonable to do in ALGOL or FORTRAN. This includes, however, only straightforward numerical computations. Attempts at hard symbol manipulation problems in ALGOL or FORTRAN generally are so much less efficient than machine code, if perhaps more readable, as to be prohibitively expensive. One may note that the reason these languages are inefficient in these sorts of computations is that the languages do not provide very much richness in their data structures or syntax - what is there is built in and effectively cast in concrete. If what one wants to do is readily done in the syntax and with the data structures provided, well and good. If not, one must look elsewhere.

There is what appears at first glance to be a solution to this problem. If ALGOL 60, for example, doesn't provide the forms appropriate to a particular problem area, surely one can find a language that does, since there are languages particularly suited to each of a large variety of problem categories. This isn't, however, always an adequate answer. If there is a language suited to a problem (and if it is a problem not like any previously treated with computer methods there probably won't be) it may well be unknown to the user who needs it. Certainly the learning of any sizeable subset of the existing programming languages is too great a burden to impose on most computer users. Even if our hypothetical user does know a language suited to his problem, he will in all likelihood discover that it has not been implemented on the computers to which he has access. Finally, and most importantly, if a particular user requires the resources of two or more distinct problem oriented languages, he will find that there is no general or easy way to obtain in a single language the desired union of subsets of several different languages.

C. General Purpose Languages

1. Shell Languages

We have a few alternatives remaining. One is to try, whenever the special purpose languages do not serve, to use one of the general purpose shell languages, e.g., PL/1. Although these do provide a very rich data space, they generally do not provide very much variability of syntax. Furthermore, they are built on the assumption that every user needs all data structures and techniques. As a consequence, they force their users to pay the overhead associated with all the language's components - those that are needed for a problem as well as those that are not. The construction of such all inclusive blends, besides being a massive programming effort, must, by its very nature, involve the making of a number of decisions at the time of language design. Even where these don't create anomalies (e.g. $7<6<5$ has the value TRUE in PL/1) many are bound to differ from the decision that would have been made by some class of future users. For example, if A and B are conformable matrices, PL/1 will always interpret A*B as the element by element product. There is no reasonable way to override this and have A*B mean the normal matrix product.

2. Extensible Languages

If then, there is a never ending source of problems for which existing special purpose and shell languages are not suited, we will be forced to build a new language processor for each of a number of machines every time such a problem arises. When one considers the number of existing programming languages, it becomes clear that many people have found it necessary to do just that. Since the building of a language processor usually requires a substantial investment of time and money, an investment one would prefer to make in a more direct attack on the problem one is solving, the cost of this approach makes it impractical as anything more than a stopgap if there is another option open.

Therefore we claim that what is needed is a type of language which is sufficiently flexible in its syntax and data as to be moldable into the forms required for a large class of problems at a cost much below that of producing a new language from scratch. We further claim that the core or extensible language is of this type. Before proceeding to describe such languages, and in particular CEL, we point out that the choice between the extensible and the shell language is often not clear-cut. The user of an extensible language who must extend it, for some problem, to the level of complexity of a PL/1, will probably produce a product that is a good deal less workable than PL/1. Moreover, the user whose problem area requires the use of forms available in some shell language, may find that the shell representation is sufficiently more efficient than any he can build out of the primitives of an extensible language, to make it cheaper to pay the extra overhead associated with the use of a shell language. The extensible language is not intended to best serve the needs of users with problems of these sorts. It is intended for the user for whom convenience of representation, directness and ease of use and variability of syntax are more precious commodities than efficiency of execution.

SECTION III

Features of Extensible Languages

A. Data

The object of the extensible language is the provision of a variable and flexible syntax and a data space that will host a very large variety of problem types. The designs of the extensible languages now extant suggest that a sufficiently rich data space is obtained by adding to the data types of the conventional languages a few data type definition operators. These are, in general, operators which act on existing data types to produce new data types. Let us now describe these operators and the definition process.

One begins with the set of atomic types initially defined in the base language. These are usually those types which, though definable by means of the data type definition operators, require special treatment for purposes of efficiency and are likely to be required in a significant class of extensions of the base. Such a set might be {real, integer, literal, nil}, though it might also contain multiprecision varieties of these. Let us now list the data type constructors.

The first of these is the operator that defines row constructors. It takes as arguments a data type and a positive integer length and produces a constructor that creates rows of that length when applied to arguments of that type. For example, if I^n denotes the row of n integers 1, 2, . . . , n then I^n = row(integer, n)(1, . . . , n). Several points should be noted about this constructor. First, it is sufficient to provide a one dimensional array constructor since an array of higher dimension k can be represented as a row of k (k-1)-dimensional arrays. Second, this constructor will construct only homogeneous rows - rows whose elements all have the same type. There are good reasons, related to the efficiency of compiled code, for including this constraint in compiled languages. In CEL however, as we shall see, these are not operative. Third we note that the row data type constructor may permit the length to be missing. In this case the data type produced will be a row of dynamic length.

A second data type constructor is the *struct* (to use the terminology of BASEL). It creates data types that have several components, in general of different types. For example the type *struct*(rp:*real*, ip:*real*) has two real components, called rp and ip respectively, and is useful to model complex numbers. In general struct$(s_i:t_i)_{i=1}^{n}$ is a data type with n components, the i^{th} called s_i and of type t_i. Another instance of this is a *struct*(stack:*row*(100, *real*), level: *integer*) which might be used to model a stack of reals whose maximum depth is 100.

A third data type constructor creates references, i. e., pointers to data. We note that one useful application of the *ref* is the modeling of "call by reference" in a language that does not explicitly provide for it - since the value of a *ref* is a pointer to the datum and the value of this value is, in turn, the datum pointed to. Moreover it permits sharing - that is, the independent accessing of the same datum via different pointers.

Finally, extensible languages generally provide a _union_ constructor which acts on a set of types $\{t_i\}_{i=1}^{n}$ to produce a type t defined by (x is of type t)$\longleftrightarrow$($\exists$ i)($1 \leq i \leq n$)(x is of type t_i). For example, we can use _union_ to define "list-of-integers" as follows

list-of-integers = _union_(pair-of-integers, _nil_)

pair-of-integers = _struct_(head:_integer_, tail:list-of-integers).

We note that there is usually a type denoted by _any_(or _general_ or _free_) defined by (x is of type _any_)$\longleftrightarrow$TRUE, i. e., every datum is of this type.

Having constructed an extended set of types, we need a number of functions to transact with them. These may be classified as constructors, predicates and selectors. For each non-atomic type t we need a function "construct-t" which given an appropriate set of arguments constructs a datum of type t. We need a predicate on two arguments which for a datum x and a type t tells us whether x is of type t. Finally, we need selectors which produce from a datum its component parts.

We note that selection is trivial for rows, and is accomplished by subscripting. For a data type _struct_$(s_i:t_i)_{i=1}^{n}$ the constructor is a function $t = \{\lambda x_1 \lambda x_2 \ldots \lambda x_n.$ _struct_$(s_i:x_i)_{i=1}^{n}\}$ and the selectors are the functions f_j such that $f_j(t(x_1, \ldots, x_n)) = x_j$. It is clear that there are a number of choices to be made as to how constructed data types and their associated functions are to be designated. Let us therefore indicate the choices made in three extensible languages - Jorrand's BASEL[4, 6, 8, 16], Garwick's GPL[8, 18], and CEL.

In BASEL to define the mode complex and deal with it one might write

```
(let complex rep struct[real rp, real ip],
    z be complex in
    z = complex[1.0, 2.0]; rp of z = 4.0;... )
```

We note that here the data type name itself is the name of the constructor function and that selection is accomplished with the operator of. BASEL provides a union type operator as described above and a special predicate to test the type of a variable, e.g.,

```
(let u be union(int or bool) in
    u = 1;...
    u = TRUE;...
    when u is int then
    factorial[u] else ......)
```

BASEL has a pointer data type called loc and provides a function alloc which creates data of this type, e.g.,

```
(let i be int, j be loc int in
    i = 1;j = alloc 2;
    j-->i;i = plus[i, val j];...)
```

Here the operator val follows the pointer and the operator "-->" points the pointer at i.

A variable which is a row of k real numbers in BASEL has type row k of real, where k is a positive integer. If arrays of dynamic size are desired, any may be substituted for k in the mode descriptor..

Similarly, to declare complex in GPL one would write

```
        block complex {real rp, ip};
        complex z;z-->complex(1.0,2.0);4.0-->rp(z);... )
```

Here the data type's name is the name of the constructor function. The '-->' is GPL's assignment operator. Selection is denoted by functional notation. Although GPL does not explicitly provide a union operator, its pointers, which have mode ptr , can point to data of variable type. A predicate much like that in BASEL is provided for type testing, e.g., one might write

```
    iff integer u ...; iff boolean u;...
```

One defines an array data type in GPL by writing, for example, "array vector of real." This defines the type vector to mean a linear array of real numbers, each accessed by specifying its ordinal position in the array. GPL permits the bounds of such an array either to be specified by declaration or determined dynamically.

The nearest equivalents of these examples in CEL have somewhat different behavioral properties because CEL is an interpretive language. This means that the machinery of the language processor is all present at the time of program execution and hence that it is possible to dispense with type declarations. The type of a variable is simply the type of the last datum assigned to it or NIL if no assignment to it has occurred. Another way to view this is to think of type as a property not of variables but of values. To make Z the complex 1.0+2.0I one would write, without preliminaries,

```
Z<--MKSTR(RP:1.0,IP:2.0).
```

The function MKSTR is one of CEL's library functions and creates a datum of type <u>struct</u>. If one wishes to create data of type complex more conveniently, one can define a function COMPLEX, taking two arguments and returning a datum constructed as above. Then one could write, as in GPL or BASEL,

```
Z<--COMPLEX(1.0,2.0).
```

Having defined the type complex, one might choose to model quaternions using the same constructor, giving it not real but instead complex arguments. Since types are dynamic, one could write

```
Z<--COMPLEX(COMPLEX(1.0,2.5),COMPLEX(3.1,-4.7)).
```

Selection from a struct is accomplished by writing the name of the component desired in square brackets following the expression whose value is the struct. For example if Z is a <u>struct</u>(RP:1.0,IP:-2.5) then Z[RP] = 1.0 and Z[IP] = -2.5. Selection can be iterated for <u>struct</u>'s with components that are <u>struct</u>'s, e.g., If a is struct(a1:<u>struct</u>(a2:struct(a3:x))) then a[a1;a2;a3] = x. Selection can also be accomplished by subscripting with the ordinal position of the component desired, as in GPL. For example, a(1)(1)(1) = x.

Predicates to test the type of a variable are available in several varieties. First of all there is a library function ILK which returns the type of its argument as an integer code (see Appendix A). Further discrimination is possible via the invocation of library functions that return the number of components of a <u>struct</u> or the name of its i^th component as a literal. One may more conveniently test the type of a datum by defining the constructor in such a way that it will insert the additional information at the time of construction, for example

```
Z  <-- MKSTR (TYPE:"COMP", RP:1. 0, IP:3. 2)

Z2<-- MKSTR (TYPE:"POLAR", RHO:1. 0, THETA:3. 2)
```

We note that constructor functions defined to create data of type Z1 or Z2 would still take two arguments. We claim that this method of detecting type is probably optimal, since any given user of the language will want to distinguish between only a subset of the data structures he is using, and he can insert the minimum amount of additional information into the data structures that is needed for this purpose.

It should be noted that the dynamic types of CEL, as opposed to the declared and fixed types of languages such as BASEL or GPL, are not always an advantage. In particular, if types are dynamic, it is necessary to execute programs interpretively so that type testing and switching are done correctly. The BASEL or GPL processors, on the other hand, will refuse to compile a statement of the form 'rp <u>of</u> z = <u>alloc</u> 1. 0' if z is complex. They can thus virtually eliminate run time type switching (exceptions being the <u>iff</u> clause of GPL and the when clause of BASEL) whereas it is always present in CEL. The tradeoff here is the standard one between efficiency and flexibility.

The notion of union in GPL, BASEL and other extensible languages is primarily a means of overcoming the restrictions imposed on the values of variables by type declarations. Since CEL contains no declarations, the union operation is largely superfluous, except where used for the convenient creation of predicates for testing type class membership as described below.

Rows are created in CEL in two ways. The first is by invoking the library function MKROW which takes as arguments any number of data and creates a row having these data as its elements. The second method of row creation is via the function MKNRW (Make Nil Row) which takes a single positive integer as argument and creates a row of that length with all its elements initially of type NIL. Since CEL types are dynamic, CEL need not make special provision for rows of dynamic size.

One property of BASEL or GPL's declarations, however, would for some purposes be a great convenience in CEL. This is that a type definition automatically creates convenient notations for the constructor and predicate functions. Indeed, let us describe a possible straightforward extension of CEL that would add to CEL a data definition facility of the sort offered in BASEL or GPL. In particular we want a facility which accepts the equivalent of BASEL's 'let complex rep struct[real rp, real ip]' and automatically adds to the system a constructor function that creates complex data and a predicate that tests whether an arbitrary datum is complex. It need not add selection functions, since a sufficient facility for these is automatically present, e.g., if z is a struct(rp:1.0, ip:2.0), then z[rp] and z[ip] (also z(1) and z(2)) are its components.

Consider the terminal sentences generated by <ddef> in the following grammar:

```
<ddef>::=$<definiendum> =<definiens>$
<definiendum>::=<identifier>
<type>::=<identifier>| real | integer| literal| any| nil
<definiens>::=<structure pattern>|<alternate pattern>|
          <sequence pattern>|<reference pattern>
<structure pattern>::=struct(<pattern component list>)
<pattern component list>::=<pattern component>|<pattern
          component>,<pattern component list>
<pattern component>::=<selector>:<type>
<selector>::=<identifier>
<alternate pattern>::=<type>|<type> or <alternate pattern>
<sequence pattern>::=seq(<type>)
<reference pattern>::=ref(<type>)
```

These are a modified version of a subset of the data definition component of Standish's Polymorphic Programming Language (PPL). The semantics associated with the rules of the above are most easily explained via some examples. Consider the following <ddef>'s:

1. Complex = struct(rp:real, ip:real)

2. Complexrow = seq(complex)

3. Listofcomplex = complex or pairofcomplex

4. Pairofcomplex = struct(car:listofcomplex, cdr:listofcomplex)

(1) and (2) define, respectively, complex variables and rows of indefinite length of complex variables. (3) and (4) define a list structure whose atoms are complex variables (for further details, see Section VI B). We desire the result of writing these definitions to be that the system generates several functions -

(a) three constructor functions - complex, complexrow and pairofcomplex which take two, indefinitely many and two arguments respectively and produce structures of the appropriate form and

(b) predicates of the form element(x, t) (which we will write as $x \epsilon t$) whose domain is $\{x | x$ is a datum$\}$ x $\{t | t$ is a <type>$\}$ which will now have the value TRUE for the following pairs of the form (datum, type): (complex, complex), (complexrow, complexrow), (complex, listofcomplex), (pairofcomplex, listofcomplex), and (pairofcomplex, pairofcomplex) as well as those pairs for which it was previously true.

We now give a precise description of the functions which are added to the set of defined functions for each <ddef>. In what follows s and s_i will denote <selector>'s and t and t_i will denote <type>'s. I(t) is a unique constant associated with the identifier t.

For each <ddef> with <definiendum> the identifier t, create a struct named t and equal to struct(predicate:a, constructor:b) with a and b defined as follows:

If the <definiens> is a <structure pattern> of the form struct$(s_i{:}t_i)_{i=1}^{n}$ set t[predicate] = $\{\lambda x.\ x[\text{type}] = i(t)\}$ and t[constructor] = $\{\lambda x_1 \ldots \lambda x_n.\ \text{MKSTR}((s_i{:}t_i)_{i=1}^{n}\ ,\text{type}{:}i(t)\ \}$

If the <definiens> is an <alternate pattern> of the form t_1 or . . . or t_n set t[predicate] = $\{\lambda x.\ (x\varepsilon t_1)$ or . . . or $(x\varepsilon t_n)\}$ and leave t[constructor] undefined.

If the <definiens> is a <sequence pattern> of the form seq(t')set t[constructor] = $\{\lambda x_1 \ \ldots\ \lambda x_n \ \text{MKROW}(x_1, \ldots, x_n)\}$ and t[predicate]= $\{\lambda x.\ x(1)\varepsilon t' \wedge \text{ILK}(x) = \text{seq}\}$. (In this and the next definition, we use 'ref' and 'seq' as variables whose value is the internal code for the types ref and seq, respectively). Here n is indeterminate.

Finally, if the <definiens> is a <reference pattern> of the form ref(t') set t[predicate] = $\{\lambda x.\ (\text{VLPTR}(x)\varepsilon t')\wedge(\text{ILK}(x) = \text{ref})\}$ and t[constructor] = $\{\lambda x.\ \text{MKREF}(x)\}$.

Define the construct and element functions by:

Construct$((x_i)_{i=1}^{n}, t)$ = if atomic(t) then undefined else t[constructor]$(x_i)_{i=1}^{n}$;Xεt = if atomic(t) then (ILK(x) = i(t)) else t[predicate](x);

B. Syntax

Much of the effectiveness that is the goal of the extensible language is lost if, although data types are flexible, syntax is not. A programmer who has defined a set of unusual data types for a particular application will probably want to program in a notation similarly selected for that application. If, for example, he has defined a data space that contains list structured objects, he may wish to decree that 'x+y' is to have the value obtained by concatenating x and y whenever either is a list, and the previously defined value in all other cases. If his application is such that he must frequently write the equivalent of the special case "for i = 1 step 1 until n do . . . " of the ALGOL 60 for statement, he may wish to specify that that is the meaning of "for i --> n do . . . ". In short, he wishes to define a syntax which emphasizes what is variable in his application, minimizes what is constant and mirrors the laws of combination and growth of the objects he is manipulating.

BASEL provides no such facility, but it is intended to be part of a larger "extensible language facility" which would presumably permit some kind of syntax variability[6]. GPL contains three methods for achieving a flexible syntax. First, it permits the definition of new infix operators with associated priorities. Second, it allows a much more general form of procedure call than is conventional, e.g., permitting a user to define "ifmid a of b, c then d else e" as the calling sequence for a procedure ifmid on five arguments. Finally, GPL contains a macro expansion facility whereby after the declaration

```
procedure dist(a, b);
iff 2 space a, b take macro sqrt((x(a)-x(b))↑2+(y(a)-y(b))↑2);
```

an occurrence of "dist(s, t)" will be expanded to produce the in line code "sqrt((x(s)-x(t))↑2+(y(s)-y(t))↑2)".

A somewhat more general and powerful facility for adding syntax variability employs the mechanism of the Brooker and Morris Compiler-Compiler. Here one specifies an augmented BNF grammar for the language in which one wants to program. The augments can, for example, be transduction elements which translate a user's extension of the base syntax into the system's base language[3, 30].

The component of CEL which provides syntax variability is at present ad hoc, and we intend eventually to provide a facility of the Brooker and Morris sort in CEL. In the current definition, a user provides a function, written in CEL, which when invoked will translate a statement in the language in which the user wishes to program to the equivalent CEL base statement. That it is possible to write functions in CEL which act as syntax transducers is demonstrated by some of the examples of CEL programs given below. However it is equally clear that one does not, in general, want to require a CEL user to program his own transduction algorithm, and hence we will replace this mechanism by one of greater sophistication in a future revised definition of CEL.

C. Control

Programming languages have several aspects whose systematic variation we may profitably study. Thus far we have discussed the data and syntax of extensible languages. One might also want to vary the control structure of a language, e. g. to obtain co-routines, clock driven simulations, multiple parallel returns by subroutines, parallel processing, continuously evaluating expressions, and so on. Thus one would expect an ideal extensible language to provide mechanisms for varying control that were sufficient to add such features. Unfortunately, determining what constitutes an optimal (or even a good) set of primitives from which common control structures can be built, is at present an unsolved problem. We know of no existing implementation of an extensible language which provides such features (with the possible exception of ALGOL 68's parallel execution expression and PL/I's ON statement) although Standish's design of PPL does make a number of such provisions which depart from orthodox control structures[29, 31]. Introducing such facilities into CEL would probably require a drastic departure from the current implementation's scheme of driving program execution from a pair of push down stacks. Hence we leave the problem of designing and implementing mechanisms for including variability of control structures in higher level languages to future researchers.

SECTION IV

Conversation and Interpretation

IV. Conversation and Interpretation

Let us now digress temporarily from the subject of extensibility to discuss the styles of debugging typical of each of two broad classes of languages, the non-interactive and interactive. The first of these consists of languages which are usually compiled and executed in a 'batch' environment. Most implementations of PL/1, ALGOL 60, FORTRAN and COBOL are in this category. On the other hand, there are interactive languages which are usually executed interpretively in a form close to that in which the programmer wrote. These languages permit the user to interact with a running program and to compose, modify and debug programs at a rapid pace. We generally find these in a time-shared and conversational environment - such languages as APL, JOSS, CAL, LISP, and so on.

Detecting errors in a non-interactive environment is usually a time consuming and tedious task. Typically one submits a program, for example as a deck of cards, and several hours later gets it back along with the results of the run. Usually these results are some mixture of wrong results and error messages. At this point, since one cannot interactively control and modify the program's execution, one can track down the source of errors only by

(a) desk checking - carrying out parts of the computation by hand, as the program directs, with sample inputs or

(b) requesting periodic printouts of partial results during the program's next run.

One generally employs some combination of these and gets, at the end of the second and successive runs, massive amounts of material, most of it irrelevant to the problem of error detection, which must nevertheless be scanned through in an attempt to find the significant parts. If it is the case that one can learn nothing at all from the results produced after the first couple of errors occurred, then one is faced with the necessity of iterating this procedure several times in order to detect bugs in a non-trivial program.

Debugging in languages of the second kind is a much less unpleasant task. Because the environment is interactive, one can dispense with core dumps and instead selectively investigate relevant evidence of errors. Good interactive languages permit one to study partial results and to phrase questions at the level of the source program. In addition one can desk check far more easily than in a batch system, because one can use the computer to do the mechanical parts (e. g. hand computation) of desk checking.

In most conversational language systems there are a number of features that further ease the debugging process. APL, for example, a prime example of a well-constructed conversational language system, allows the programmer to set break points in the program which, when encountered during execution, will suspend execution and return control to the user's teletype. He can then examine and modify the values of variables before continuing the computation at the break or any other point. APL further permits the user to trace the execution of a subset of program statements, i.e., to specify that every time one of these statements has been executed, the number of the statement and its result are to be output on the user's teletype. Once an error has been located by these means, a user may edit single statements or larger parts of the program, and immediately investigate the effect of the changes made[1, 15]. Finally, we note that in a conversational system one can take advantage of strange occurrences (e.g., a computation taking longer than it should or an output that differs from expectations) to look at the effect of an error near its source. In batch systems, by comparison, one frequently detects the presence of an error via some subtle change at a remote place in the program or its output.

Lest we be accused of unfairly stating the relative merits of interactive and non-interactive systems, we hasten to point out that experimental comparisons of productivity in these two media are not conclusive. It is difficult to obtain good comparisons between programmers working in different languages and at the same time it is unfair to draw conclusions from the performance of a batch language in a time-shared system or vice-versa. Thus we can only state our own distinct preference for conversational systems and point to the similar statements and results of others[14, 24, 27].

It is certainly the case that interpreted programs do not run as rapidly as compiled versions of the same programs. However, in many applications such factors as the programmer's ability to absorb results acts as a more restrictive constraint than running time. Moreover, one can often achieve the best results of both compilation and interpretation by including, in an interpreted language, a compile operator (e. g. the one in LISP) which one applies to a function after one is through debugging it. It is converted into machine code and thereafter runs at the rate of a compiled program. Finally there is evidence (qv. [14]) that total costs of man hours and machine time are lower in time-shared systems.

We claim that the dynamic style of programming in a conversational system meshes very nicely with the flexible nature of extensible languages. The resulting freedom is a major step toward making the process of programming as natural as possible for the programmer. This is consistent with the philosophical objective, in extensible languages, of letting a programmer express his thoughts about a problem with a minimum of artifice or translation.

The pressure of time has prevented the inclusion in CEL's current implementation of a full-fledged conversational debugging facility. One can do automated desk checking via an immediate execution feature, but this would not be adequate in the final form of the language. We hope to add a break point debugging and selective trace facility such as that of APL.

The current implementation does not contain, again because of the pressures of time, a text editing component. Until such a component can be implemented, an ad hoc provision has been made, whereby CEL programs can be created and edited using the PDP-10's TECO (Text Editor and Corrector) Program. This measure has sufficed for the creation of the CEL programs given below.

SECTION V

Description of CEL

Let us now describe CEL in some detail. The fragments of CEL programs given above are written in the language defined by the standard front end syntax of the current implementation. However, for the purposes of this section, we revert to thinking in terms of CEL's base language. This is defined by the following grammar with root symbol <program>:

```
1         <program>::=<message>
2-3       <message>::=<function definition>|<statement>
4         <function definition>::=$<function header><function body>$
5         <function header>::=<identifier>(<identifier list>)<identifier
                    list>;<identifier list>;
6-7       <identifier list>::= empty|<identifier>{,<identifier>}*
8         <function body>::={<statement>}+
9         <statement>::={<expression>}+
10-14     <expression>::=<identifier>|<constant>|({<expression>}+)
                     <expression>:<identifier>|<expression>}<identifier>
```

In this grammar, ::=, | , { , } , + and * are metasymbols. {A}* means zero or more A's. $\{A\}^+$ means one or more A's. <identifier> and <constant> are lexical tokens whose composition need not further concern us here.

A CEL program is a sequence of either function definitions or direct commands. The former result in the saving of the defined function for later execution, whereas the latter usually invoke one or more previously defined functions. We see from (9) that a statement is a sequence of expressions - the associated meaning is that the execution of a statement consists of the successive evaluation of its constituent expressions, read from left to right. The value of an expression e is recursively defined as follows (the value assigned to <identifier> is later qualified):

```
Val(e) = if e is a constant then e else
         if e is an identifier then if an assignment operator
            has been called with the identifier as the left
            operand, then the value of the right operand at
            the last such call within the scope of the identifier
            and NIL otherwise else
         if e is of the form e']i then
            if e' is a struct with a component named i then val
               applied to the i component of e' and otherwise error
               else
         if e is of the form e':i then val(e') else
         if e is of the form (e_1 e_2 ... e_n e') then
            if e' is a row then
               if n = 1 and val(e_1) is a positive integer m
                  < length(e') then val applied to the mth component
                  of e' and otherwise error else
            if e' is a function then (see below) else
         error;
```

We must now define the value of the expression $e = (e_1 \ldots e_n e')$ where e' is a function. If n is not the number of arguments that e' requires, there is an error condition. If e' is a library function, the value and/or side effects due to evaluating it with arguments $e_1, \ldots, e_n$ are specified in Appendix A. Otherwise e' is a programmer defined function. E is then evaluated by calling e', supplying it with $\mathrm{val}(e_1), \ldots, \mathrm{val}(e_n)$ as arguments and using the result(s) returned by e' at its exit as val(e).

Finally we define the process of calling a programmer defined function f with arguments $e_1, \ldots, e_n$. Suppose the header of f is "$f(x_1, \ldots, x_s)r_1, \ldots, r_t; l_1, \ldots, l_u;$" and that it contains k statements, numbered $1, \ldots, k$. Then to call f we

(1) Set a program counter c to one.

(2) Execute the c'th statement of f.

(3) Set c to c+1. If $c > k$ then exit returning $r_1, \ldots, r_t$ as results. Otherwise, proceed from (2).
(A transfer statement achieves its effect by changing the value of c).

An identifier i is evaluated within f as follows - if i is not one of the locals of f (i.e. one of the x_i's, r_i's or l_i's) then its value is the same as if i had been encountered outside f. If i is a local of f, its value is that last assigned to i since f was last entered. If no such assignment has been made and i is not one of the x_i's, then val(i) = NIL. If i is x_j, then $\mathrm{val}(i) = \mathrm{val}(e_j)$. An assignment within f to a local of f has no effect on the value of another identifier of the same name existing outside f. Here "outside f" means either inside some other function, inside any other call of f (if f is recursive) or outside all functions.

This base language has been chosen for CEL for several reasons. It is relatively simple yet powerful enough to express all the constructs we wish to include in CEL. Statements in it can be executed efficiently since it is SLR(1) (i. e. , at any stage in a parse, the next applicable reduction is unambiguously determined by inspection of at most one symbol to the right of the current symbol) and indeed, the current implementation does a small amount of preprocessing to make it SLR(0). There are no reserved words, which makes learning the language easier than it would otherwise be. Finally, we can translate from a number of front end syntaxes to this base without much difficulty. On the other hand, programs written in the base are relatively unreadable. Because we always intend to program in some front end syntax, this is not a problem.

CEL's standard front end syntax in the current implementation is defined in terms of the base syntax by the following transduction grammar G:

<statement>::=<identifier>:<expression> $@_T$<expression>:<identifier>

<statement>::=<expression>

<expression>::=<term>α<expression> $@_T$(<term><expression>f(α))

<expression>::=α<expression> $@_T$(<expression>f(α))

<expression>::=<term>

<term>::=<constant>|<identifier>

<term>::=(<statement>) $@_T$<statement>

<term>::=<term>() $@_T$(<term>)

<term>::=<term>({statement>}$^+$) $@_T$ ({<statement>}$^+$<term>)

<term>::=<term>[<identifier>{;<identifier>}*] $@_T$<term>]<identifier>
{]<identifier>}*

There is a version of each rule containing "α" for each infix operator that is to be used. For each such operator, f(α) is the name of a CEL function that computes its value for the correct number of arguments.

The base language translation of a statement written in this syntax may be determined by following this recipe -

(1) Obtain a parse tree in G for the statement. This is a particularly easy process since G is an operator precedence grammar with f and g functions (qv. [11] and Appendix B).

(2) Rearrange the sons of each nonterminal node x (possibly deleting some and inserting others) according to the transduction element for the rule of g that corresponds to the node and its

sons.

(3) Read off the terminal nodes of the modified tree in left to right order, obtaining the translated statement.

SECTION VI

Programming in CEL

A. An Introduction

We now present several extensions of CEL. Each consists of a number of functions which manipulate the data types of the extension. In order to conveniently explain how they work, we will insert descriptive text between fragments of the actual CEL text, though of course this description would be omitted during a computer session using CEL.*

The program text was produced by a device which uses " → " for "-->" and " × " for "".

B. List Processing

The major data types of this extension are defined as follows:

Atom = integer or real or literal or nil

Pair = struct(car:listel, cdr:listel)

Listel = atom or pair

The data type pair corresponds to the dotted pair of LISP. We first define several of the elementary functions of LISP -

```
    $CONS(A,B)R;;
[1] R<--MKSTR(CAR:A,CDR:B)
    $


    $CAR(X)R;;
[1] ATOM(X)→4
[2] R<--X[CAR]
    $


    $CDR(X)R;;
[1] ATOM(X)→3
[2] R<--X[CDR]
    $


    $ATOM(X)R;;
[1] R<--ILK(X)≠4
    $
```

Cons is a constructor function that takes a pair of arguments, presumably of type listel, and makes a pair out of them. Car and cdr are generalized selectors - they return the appropriate component if it exists, but NIL if it doesn't (i. e. if the argument is an atom). They accomplish this by, for atomic arguments, exiting from the procedure without making an assignment to the result R, and hence return NIL according to the rules given in Section V. Atom is a predicate which distinguishes atoms from non-atoms using the fact that in this extension the only non-atoms are structs, and that any struct x satisfies ILK(x) = 4. We next define a function list which creates structures corresponding to the list of LISP, i.e., data x satisfying

(1) $x \in$ listel and

(2) if atom(cdr(x)) then cdr(x) = NIL.

```
    $LIST(X)R;;
[1] (ILK(X)=6)→4
[2] R<--CONS(X,NIL)
[3] →20
[4] J<--LNGTH(X)
[5] R<--CONS(X(J),R)
[6] (J=1)→20
[7] J<--J- 1
[8] →5
    $
```

Lists are a proper subset of pairs, but are often easier to deal with in that we can view car as returning the first element of the list and cdr as returning the list obtained by deleting the first element of the original list. Since we want the constructor list to be variadic, it is defined so that it will accept either a single argument or a row of arguments, i.e., list(x) = if atomic(x) then cons(x, NIL) else if ILK(x) = row then cons(x_1, cons(. . . , cons(x_n, NIL). . .) where n is the length of x.

Finally we can define a number of functions on lists - these are standard functions in LISP, implemented by means of great reliance on the idea of recursion. Append creates a list of all the elements of its arguments lists. Reverse creates a list in which the elements of the original list appear in the reverse of their original order. Showd and showp make recursive calls on each other in order to print a list showing the list structure with parentheses. Seek takes as arguments an atom and a list of pairs, and returns the list whose elements are the right halves of all pairs whose left half is the atom. Replace changes all occurrences of an atom in a list to another atom. Same determines whether two list structures are the same and member determines whether one list is a sublist of another. These last two are used by union to construct a list whose elements are those in the union of the sets of elements of the two argument lists. Finally, map applies a function to successive cdrs of a list and returns the final cdr (NIL for a list) as result.

```
      $REPLACE(A,B,L)R;;
[1]   R<--B
[2]   (L=A)→7
[3]   ATOM(L)→6
[4]   R<--CONS(REPLACE(A,B,L[CAR]),REPLACE(A,B,L[CDR]))
[5]   →7
[6]   R<--L
      $
```

```
      $SAME(S,T)R;;
[1]   ATOM(S)→5
[2]   ATOM(T)→5
[3]   R<--SAME(CAR(S),CAR(T))×SAME(CDR(S),CDR(T))
[4]   →6
[5]   R<--S=T
      $
```

```
        $SHOWD(L);;
   [1]  ATOM(L)→7
   [2]  TYPE("(")
   [3]  SHOWD(L[CAR])
   [4]  SHOWP(L[CDR])
   [5]  TYPE(")")
   [6]  →8
   [7]  TYPE(L)
        $


        $SHOWP(L);;
   [1]  (L=NIL)→7
   [2]  ATOM(L)→6
   [3]  SHOWD(CAR(L))
   [4]  SHOWP(CDR(L))
   [5]  →7
   [6]  TYPE(L)
        $


        $SHOW(L);;
   [1]  SHOWD(L)
   [2]  TYPE("
")

        $


        $SEEK(L,A)R;;
   [1]  (ILK(L)=0)→6
   [2]  (A≠CAR(CAR(L)))→4
   [3]  R<--CONS(CDR(CAR(L)),R)
   [4]  L<--CDR(L)
   [5]  →1
        $

        $REVERSE(X)R;;
   [1]  (ILK(X)=0)→3
   [2]  R<--APPEND(REVERSE(CDR(X)),CONS(CAR(X),NIL))
        $

        $LENGTH(X)R;;
   [1]  ATOM(X)→4
   [2]  R<--1+LENGTH(CDR(X))
   [3]  →5
   [4]  R<--0
        $

        $APPEND(X,Y)R;;
   [1]  R<--Y
   [2]  (ILK(X)=0)→4
   [3]  R<--CONS(CAR(X),APPEND(CDR(X),Y))
        $
```

```
     $MEMBER(S,T)R;;
[1]  ATOM(T)→4
[2]  R<--SAME(S,T)+SAME(S,T[CAR])+MEMBER(S,T[CDR])
[3]  →5
[4]  R<--S=T
     $
```

```
     $UNION(L1,L2)R;S;
[1]  (L2=NIL)→6
[2]  MEMBER(L2[CAR],L1)→4
[3]  S<--CONS(L2[CAR],S)
[4]  L2<--CDR[L2]
[5]  →1
[6]  R<--APPEND(L1,S)
     $
```

```
     $MAP(F,L)R;;
[1]  F(L)
[2]  ATOM(L)→5
[3]  L<--CDR(L)
[4]  →1
[5]  R<--L
     $
```

Some instances of output obtained during runs with this extension are as follows:

```
         P<--LIST(MKROW("A",1,"B",2))
         SHOW(P)
(A1B2)
         PP<--LIST(MKROW(P,P,P))
         SHOW(PP)
((A1B2)(A1B2)(A1B2))
         PP[CDR;CAR]<--LIST(MKROW(4,5,5))
         SHOW(PP)
((A1B2)(455)(A1B2))
         Q<--APPEND(PP,PP)
         SHOW(Q)
((A1B2)(455)(A1B2)(A1B2)(455)(A1B2))
         R<--LIST(MKROW(1,2,LIST(MKROW("A","B")),3))
         SHOW(R)
(12(AB)3)
         SHOW(REVERSE(R))
(3(AB)21)
         MAP(SHOW,R)
(12(AB)3)
(2(AB)3)
((AB)3)
(3)
```

```
        S<--LIST(MKROW(CONS("A",1),CONS("B",2),CONS("C",3)))
        SHOW(S)
((A1)(B2)(C3))
        SHOW(SEEK(S,"A"))
(1)
        S<--APPEND(S,S)
        SHOW(S)
((A1)(B2)(C3)(A1)(B2)(C3))
        SHOW(SEEK(S,"A"))
(11)
        R<--LIST(MKROW("A","B"))
        S<--LIST(MKROW("A",R,"C"))
        T<--LIST(MKROW(R,S,"A",R))
        SHOW(R)
(AB)
        SHOW(S)
(A(AB)C)
        SHOW(T)
((AB)(A(AB)C)A(AB))
        MEMBER(S,T)
1
        MEMBER(T,S)
0
        MEMBER(R,T)
2
```

C Polynomials

Having defined a definition set for lists, we can use it to model polynomials. A polynomial $\sum_{i=1}^{n} a_i x^i$ is representable as a vector $(a_i)_{i=1}^{i=n}$ but this representation is inefficient if many of the a_i's are zero. In that case we prefer to represent a polynomial as a list of terms, where

Term = <u>struct</u>(deg:<u>integer</u> , coef:<u>integer</u>).

The zero polynomial is the NIL list. The following functions then suffice to construct, output, add and multiply polynomials. Examples of the out p ut they produce are given.

```
      $ADDPOLY(X,Y)R;D1,D2,S;
[1]   (X≠NIL)→4
[2]   R<--Y
[3]   →20
[4]   (Y≠NIL)→7
[5]   R<--X
[6]   →20
[7]   D1<--CAR(X)[DEG]
[8]   D2<--CAR(Y)[DEG]
[9]   (D1≠D2)→16
[10]  S<--CAR(X)[COEF]+CAR(Y)[COEF]
[11]  (S≠0)→14
[12]  R<--ADDPOLY(CDR(X),CDR(Y))
[13]  →20
[14]  R<--CONS(MKSTR(DEG:D1,COEF:S),ADDPOLY(CDR(X),CDR(Y)))
[15]  →20
[16]  (D1-D2)→19
[17]  R<--CONS(CAR(Y),ADDPOLY(X,CDR(Y)))
[18]  →20
[19]  R<--CONS(CAR(X),ADDPOLY(CDR(X),Y))
      $
```

```
        $SHOWPOLY(X);L;
   [1]  (X=NIL)→17
   [2]  L<--CAR(X)
   [3]  ((L[COEF]=1)×L[DEG]≠0)→5
   [4]  TYPE(L[COEF])
   [5]  (L[DEG]=0)→10
   [6]  TYPE("X")
   [7]  (L[DEG]=1)→10
   [8]  TYPE("↑")
   [9]  TYPE(L[DEG])
   [10] (CDR(X)=NIL)→17
   [11] X<--CDR(X)
   [12] CAR(X)[COEF]→14
   [13] →2
   [14] TYPE("+")
   [15] →2
   [16] TYPE("0")
   [17] TYPE("
")
        $


        $POLY(R)P;J;
   [1]  J<--LNGTH(R)
   [2]  P<--CONS(MKSTR(DEG:R(J),COEF:R(J- 1)),P)
   [3]  J<--J- 2
   [4]  J→2
        $


        $MULPOLY(P1,P2)R;;
   [1]  ((P1=NIL)+P2=NIL)→3
   [2]  R<--ADDPOLY(DIST(CAR(P1),P2),MULPOLY(CDR(P1),P2))
        $


        $DIST(T,P)R;L1,L2;
   [1]  (P=NIL)→5
   [2]  L1<--T[DEG]+P[CAR;DEG]
   [3]  L2<--T[COEF]×P[CAR;COEF]
   [4]  R<--CONS(MKSTR(DEG:L1,COEF:L2),DIST(T,P[CDR]))
        $
        Y<--POLY(MKROW(1,2,3,1))
        SHOWPOLY(Y)
X↑2+3X
        SHOWPOLY(MULPOLY(Y,Y))
X↑4+6X↑3+9X↑2
        Z<--POLY(MKROW(3,5,-1,2,5,0))
        SHOWPOLY(Z)
3X↑5-1X↑2+5
        SHOWPOLY(ADDPOLY(Y,Z))
3X↑5+3X+5
```

D Formulas

In the next extension we will manipulate formulas, i. e., data defined by the following definitions:

Form = <u>struct(</u>lp:formula, op:<u>literal</u>, rp:formula)

Formula = form <u>or</u> atom

Atom = <u>literal</u> <u>or</u> <u>real</u>.

First we define several functions which perform arithmetic operations on formulas and do some simplification (using the identities 1*x = x*1 = x, 0+x = x+0 = x and 0*x = x*0 = 0). The functions named SADD, SMUL, SSUB and SDIV are those invoked by the infix binary operators +, *, -. and /, respectively. They normally are functions that take any combination of integer and real arguments, but here we redefine them as follows:

```
      REAL<--1
      VARIABLE<--3
      STRUCT<--4

      $FORMULA(X,Y,Z)R;;
[1]   R<--MKSTR(LP:X,OP:Y,RP:Z)
      $


      $SADD(X,Y)Z;;
[1]   (X=0.0)→11
[2]   (Y=0.0)→9
[3]   (ILK(X)=REAL)→6
[4]   Z<--FORMULA(X,"+",Y)
[5]   →12
[6]   (ILK(Y)≠REAL)→4
[7]   Z<--RPLUS(X,Y)
[8]   →12
[9]   Z<--X
[10]  →12
[11]  Z<--Y
      $
      $SSUB(X,Y)Z;;
[1]   (Y=0.0)→5
[2]   (ILK(X)=REAL)→7
[3]   Z<--FORMULA(X,"-",Y)
[4]   →9
[5]   Z<--X
[6]   →9
[7]   (ILK(Y)≠REAL)→3
[8]   Z<--RSUB(X,Y)
      $

      $SMUL(X,Y)Z;;
[1]   (Y=1.0)→12
[2]   (X=1.0)→10
[3]   (X=0.0)→8
[4]   (Y=0.0)→8
[5]   (ILK(X)=REAL)→14
[6]   Z<--FORMULA(X,"×",Y)
[7]   →16
[8]   Z<--0.0
[9]   →16
[10]  Z<--Y
[11]  →16
[12]  Z<--X
[13]  →16
[14]  (ILK(Y)≠REAL)→6
[15]  Z<--RMULT(X,Y)
      $

      $SDIV(X,Y)Z;;
[1]   (Y=1.0)→8
[2]   (X=0.0)→6
[3]   (ILK(X)=REAL)→10
[4]   Z<--FORMULA(X,"/",Y)
[5]   →12
[6]   Z<--0.0
[7]   →12
[8]   Z<--X
[9]   →12
[10]  (ILK(Y)≠REAL)→4
[11]  Z<--RDIV(X,Y)
      $
```

We now define recursive functions which output formulas, differentiate them and substitute formulas for variables in other formulas. Examples of output follow the definitions.

```
          $PRINT(X);;
     [1]  (ILK(X)=STRUCT)→4
     [2]  TYPE(X)
     [3]  →9
     [4]  TYPE("(")
     [5]  PRINT(X[LP])
     [6]  TYPE(X[OP])
     [7]  PRINT(X[RP])
     [8]  TYPE(")")
          $
```

```
          $SHOW(X);;
     [1]  PRINT(X)
     [2]  TYPE("
")
          $
```

```
          $DERIV(E,X)R;UDASH,VDASH;
     [1]  (ILK(X)≠VARIABLE)→23
     [2]  (E=X)→20
     [3]  (ILK(E)=STRUCT)→5
     [4]  →22
     [5]  UDASH<--DERIV(E[LP],X)
     [6]  VDASH<--DERIV(E[RP],X)
     [7]  (E[OP]="+")→12
     [8]  (E[OP]="-")→14
     [9]  (E[OP]="/")→16
     [10] (E[OP]="×")→18
     [11] →23
     [12] R<--UDASH+VDASH
     [13] →23
     [14] R<--UDASH-VDASH
     [15] →23
     [16] R<--((UDASH×E[RP])-VDASH×E[LP])/E[RP]×E[RP]
     [17] →23
     [18] R<--(UDASH×E[RP])+VDASH×E[LP]
     [19] →23
     [20] R<--1.0
     [21] →23
     [22] R<--0.0
          $
```

```
        $SUBST(E,X,A)R;;
   [1]  (ILK(X)≠VARIABLE)→9
   [2]  (ILK(E)≠STRUCT)→5
   [3]  R<--FORMULA(SUBST(E[LP],X,A),E[OP],SUBST(E[RP],X,A))
   [4]  →9
   [5]  (E=X)→8
   [6]  R<--E
   [7]  →9
   [8]  R<--A
        $
```

```
        F<--("A"+"X")/("B"+"X")
        SHOW(DERIV(F,"X"))
(((B+X)-(A+X))/((B+X)×(B+X)))
        Q<--F×F
        SHOW(Q)
(((A+X)/(B+X))×((A+X)/(B+X)))
        SHOW(DERIV(Q,"X"))
(((((B+X)-(A+X))/((B+X)×(B+X)))×((A+X)/(B+X)))+((((B+X)-(A+X))/((B+X)×
(B+X)))×((A+X)/(B+X))))
        SHOW(SUBST(F,"A",F))
((((A+X)/(B+X))+X)/(B+X))
```

E Vectors

In this extension we define functions that do arithmetic with 3-vectors, where

3-vector = <u>struct</u>(i:arith, j:arith, k:arith)

Arith = <u>real</u> <u>or</u> <u>integer</u>.

The infix arithmetic operators are redefined to accept arguments that are any combination of arith and 3-vector and to produce appropriate results. Par and perp are predicates that test whether a pair of vectors are, respectively, parallel or perpendicular. The definitions and some results are as follows:

```
     $VECTOR(A,B,C)R;;
[1]  R<--MKSTR(I:A,J:B,K:C)
     $

     ADD<--SADD
     SUB<--SSUB
     MUL<--SMUL
     EQ<--EQUAL

     $SADD(X,Y)R;;
[1]  R<--OP(X,Y,ADD)
     $


     $SSUB(X,Y)R;;
[1]  R<--OP(X,Y,SUB)
     $


     $OP(X,Y,Z)R;L;
[1]  (ILK(X)≠4)→7
[2]  (ILK(Y)≠4)→5
[3]  R<--VECTOR(Z(X[I],Y[I]),Z(X[J],Y[J]),Z(X[K],Y[K]))
[4]  →11
[5]  Y<--VECTOR(Y,Y,Y)
[6]  →3
[7]  (ILK(Y)≠4)→10
[8]  X<--VECTOR(X,X,X)
[9]  →3
[10] R<--Z(X,Y)
     $
```

```
        $EQUAL(X,Y)R;;
   [1]  (ILK(X)≠4)→5
   [2]  (ILK(Y)≠4)→5
   [3]  R<--(X[I]=Y[I])×(X[J]=Y[J])×(X[K]=Y[K])
   [4]  →6
   [5]  R<--EQ(X,Y)
        $


        $SMUL(X,Y)R;C1,C2,C3;
   [1]  (ILK(X)≠4)→10
   [2]  (IK(Y)≠4)→8
   [3]  C1<--(X[J]×Y[K])-X[K]×Y[J]
   [4]  C2<--(X[K]×Y[I])-X[I]×Y[K]
   [5]  C3<--(X[I]×Y[J])-X[J]×Y[I]
   [6]  R<--VECTOR(C1,C2,C3)
   [7]  →14
   [8]  R<--VECTOR(X[I]×Y,X[J]×Y,X[K]×Y)
   [9]  →14
   [10] (ILK(Y)≠4)→13
   [11] R<--VECTOR(X×Y[I],X×Y[J],X×Y[K])
   [12] →14
   [13] R<--MUL(X,Y)
        $


        $SHOW(X);;
   [1]  TYPE(X[I])
   [2]  TYPE("I")
   [3]  IFLJM(5,X[J])
   [4]  TYPE("+")
   [5]  TYPE(X[J])
   [6]  TYPE("J")
   [7]  IFLJM(9,X[K])
   [8]  TYPE("+")
   [9]  TYPE(X[K])
   [10] TYPE("K")
   [11] TYPE("
")
        $


        $DOT(X,Y)R;;
   [1]  R<--(X[I]×Y[I])+(X[J]×Y[J])+X[K]×Y[K]
        $


        $PERP(X,Y)R;;
   [1]  R<--ZERO(DOT(X,Y))
        $
        $PAR(X,Y)R;L;
    [1] L<--X×Y
    [2] R<--ZERO(L[I])×ZERO(L[J])×ZERO(L[K])
        $


        $ZERO(X)R;;
    [1] R<--(X=0)+(X=0.0)
        $



        R<--VECTOR(1,2,5)
        S<--VECTOR(2,-3,7)
        SHOW(R)
1I+2J+5K
        SHOW(S)
2I-3J+7K
        SHOW(R+S)
3I-1J+12K
        SHOW(R×S)
29I+3J-7K
        DOT(R,S)
31
        PERP(R,S)
0
        PAR(R,S)
0
        T<--2×R
        SHOW(T)
2I+4J+10K
        PAR(R,T)
1
        PERP(R,R×S)
1
        SHOW(4+R)
5I+6J+9K
        SHOW(R+3)
4I+5J+8K
```

F Trees and Syntax

In this extension we create trees and use them in various syntax manipulations. A tree is defined by

Tree = struct(father:ptr, rightbrother:ptr, firstson:ptr, value:atom)

Ptr = NIL or pointer

Pointer = ref(tree)

Atom = literal or integer or real or NIL.

First we redefine the infix binary operators so that they will create trees, i.e., so that a α b (where α is a binary operator) is a tree whose value is α, whose rightbrother and father are NIL, and whose firstson is a tree with value = a, father = a α b, firstson = NIL, and rightbrother = a tree with value = b, rightbrother and firstson = NIL, and father = a α b. This is an instance of the use of ref's to share data since the node with value α is shared as father by each of its sons. This representation of a tree with three links associated with each node is that suggested by Cheatham in [7].

```
     $SADD(X,Y)R;;
[1]  R<--OPER(X,Y,"+")
     $

     $SSUB(X,Y)R;;
[1]  R<--OPER(X,Y,"-")
     $

     $SMUL(X,Y)R;;
[1]  R<--OPER(X,Y,"×")
     $

     $SDIV(X,Y)R;;
[1]  R<--OPER(X,Y,"/")
     $
     $OPER(A,B,C)R;;
[1]  (ILK(A)≠4)→8
[2]  (ILK(B)≠4)→10
[3]  A[RIGHTBROTHER]<--MKREF(B)
[4]  R<--MKSTR(FATHER:NIL,RIGHTBROTHER:NIL,FIRSTSON:MKREF(A),VALUE:C)
[5]  A[FATHER]<--MKREF(R)
[6]  B[FATHER]<--MKREF(R)
[7]  →12
[8]  A<--MKSTR(FATHER:NIL,RIGHTBROTHER:NIL,FIRSTSON:NIL,VALUE:A)
[9]  →2
[10] B<--MKSTR(FATHER:NIL,RIGHTBROTHER:NIL,FIRSTSON:NIL,VALUE:B)
[11] →3
     $
```

The problem of how to output the constructed structure is somewhat more difficult in this than in the previous extensions given, particularly since a tree is an inherently two-dimensional object and we wish to display it in a linear medium. There are several standard ways of doing this, all involving the traversal of the nodes in some specified order. We show three typical ones. In prefix walk order we start at the roots of the tree and at each step go to

(1) the firstson of the current node if there is one or

(2) the rightbrother if there is no firstson (in general, the right neighbor - the rightbrother of the closest ancestor who has a rightbrother).

In suffix walk order, we traverse terminal nodes in left to right order, always traversing an interior node as soon as we have encountered all of its sons. Finally, in constant depth walk order, we traverse first the root, then all sons at depth two, and so on. The prefix and suffix functions use explicit recursion to traverse subtrees, whereas the constant depth function uses a push down stack, i. e. , a datum defined by

Stack = struct(stk:row100, level:integer)

Row100 = seq (100).

Output obtained with these functions follows their definitions.

```
        $SHOWPREFIX(X);;
    [1] SHOWP(X)
    [2] TYPE("
")
        $

        $SHOWP(X);L;
    [1] TYPE(X[VALUE])
    [2] (X[FIRSTSON]=NIL)→11
    [3] TYPE("(")
    [4] L<--VLPTR(X[FIRSTSON])
    [5] SHOWP(L)
    [6] (L[RIGHTBROTHER]=NIL)→10
    [7] TYPE(",")
    [8] L<--VLPTR(L[RIGHTBROTHER])
    [9] →5
    [10] TYPE(")")
        $

        $SHOWS(A);L;
    [1] (A[FIRSTSON]=NIL)→10
    [2] TYPE("(")
    [3] L<--VLPTR(A[FIRSTSON])
    [4] SHOWS[L)
    [5] (L[RIGHTBROTHER]=NIL)→9
    [6] TYPE(",")
    [7] L<--VLPTR(L[RIGHTBROTHER])
    [8] →4
    [9] TYPE(")")
    [10] TYPE(A[VALUE])
        $

        $SHOWSUFFIX(A);;
    [1] SHOWS(A)
    [2] TYPE("
")
        $
```

```
         $SHOWCD(A);X,I;
    [1]  S1<--MKSTR(STK:MKNRW(50),LEVEL:0)
    [2]  PUSH(MKREF(S1),MKREF(A))
    [3]  S2<--MKSTR(STK:MKNRW(50),LEVEL:0)
    [4]  ASIGN(I,1,TYPE("("))
    [5]  X<--VLPTR(S1[STK](I))
    [6]  TYPE(" ")
    [7]  TYPE(X[VALUE])
    [8]  (X[FIRSTSON]=NIL)→10
    [9]  PUSH(MKREF(S2),X[FIRSTSON])
    [10] (X[RIGHTBROTHER]=NIL)→13
    [11] X<--VLPTR(X[RIGHTBROTHER])
    [12] →6
    [13] (I=S1[LEVEL])→16
    [14] I<--IPLUS(I,1)
    [15] →5
    [16] TYPE(")")
    [17] (S2[LEVEL]=0)→21
    [18] S1<--S2
    [19] S2[LEVEL]<--0
    [20] →4
         $


         $SHOWCONSTANTDEPTH(A);;
    [1]  SHOWCD(A)
    [2]  TYPE("
")
         $



         $PUSH(X,Y);;
    [1]  VLPTR(X)[LEVEL]<--VLPTR(X)[LEVEL]+1
    [2]  VLPTR(X)[STK](VLPTR(X)[LEVEL])<--Y
         $
```

```
        Y<--(3+7)-(4×8)/5
        SHOWPREFIX(Y)
-(+(3,7),/(×(4,8),5))
        SHOWSUFFIX(Y)
((3,7)+,((4,8)×,5)/)-
        SHOWCONSTANTDEPTH(Y)
( -)( + /)( 3 7 × 5)( 4 8)
        A<--Y+Y
        SHOWPREFIX(A)
+(-(+(3,7),/(×(4,8),5)),-(+(3,7),/(×(4,8),5)))
        SHOWCONSTANTDEPTH(A)
( +)( - -)(+ / + /)( 3 7 × 5 3 7 × 5)( 4 8 4 8)
        SHOWSUFFIX(A)
(((3,7)+,((4,8)×,5)/)-,((3,7)+,((4,8)×,5)/)-)+
```

A major application of the data type tree occurs in the problem of parsing a sentence generated by a general context free grammar. Rather than show functions which solve this general problem, we develop routines which parse a sentence in a context free grammar which is simple precedence with f and g functions (qv. [7]). The variable grammar is a representation of a simple precedence grammar, namely

S ::= E RPAD

E ::= E+T

E ::= T

T ::= A

T ::= AT

The first rule is not included in the representation, since it is recognized "by hand" by the functions we define. Parse employs a push down stack to save the fragments of the parse tree constructed at some stage in a parse, and uses the f and g functions to decide when to make a reduction. Match makes reductions, recognizing the rule to be used, and growing the appropriate piece of tree. When parse sees the right pad symbol, it constructs the final tree fragment - the root node, and returns this as result. The functions and some typical output are:

```
      $PARSE(X,GR,F,G)R;Z,RR,X1,I;
[1]   Z<--MKSTR(STK:MKNRV(50),LEVEL:0)
[2]   RR<--MKREF(Z)
[3]   ASGNC(X1,VLPTR(X))
[4]   PUSH(RR,MKREF(NODE(NIL,NIL,NIL,X1(1))))
[5]   I<--2
[6]   (F(VLPTR(Z[STK](Z[LEVEL]))[VALUE])-G(X1(I)))→11
[7]   PUSH(RR,MKREF(NODE(NIL,NIL,NIL,X1(I))))
[8]   (X1(I)="RPAD")→13
[9]   I<--I+1
[10]  →6
[11]  MATCH(RR,GR)
[12]  →6
[13]  I<--1
[14]  VLPTR(Z[STK](I))[RIGHTBROTHER]<--Z[STK](I+1)
[15]  VLPTR(Z[STK](I))[FATHER]<--MKREF(R)
[16]  I<--I+1
[17]  (I≠Z[LEVEL])→14
[18]  R<--NODE(NIL,NIL,Z[STK](1),"S")
[19]  VLPTR(Z[STK](I))[FATHER]<--MKREF(R)
      $


      $MATCH(RR,GR);I,M,Z,J,K,N,S,T;
[1]   I<--1
[2]   ASGNC(T,VLPTR(GR))
[3]   ASGNC(Z,VLPTR(RR))
[4]   J<--LNGTH(T(I)[RP])
[5]   K<--J
[6]   N<--Z[LEVEL]+K-J
[7]   N→10
[8]   I<--I+1
[9]   →4
[10]  (T(I)[RP](K)=VLPTR(Z[STK](N))[VALUE])→12
[11]  →8
[12]  K<--K- 1
[13]  K→6
[14]  EQUAL(M<--N,M,Z[LEVEL])→19
[15]  VLPTR(Z[STK](M))[FATHER]<--MKREF(S)
[16]  VLPTR(Z[STK](M))[RIGHTBROTHER]<--Z[STK](M+1)
[17]  M<--M+1
[18]  (M≠Z[LEVEL])→15
[19]  VLPTR(Z[STK](M))[FATHER]<--MKREF(S)
[20]  Z[LEVEL]<--N
[21]  S<--NODE(NIL,NIL,Z[STK](N),T(I)[LP])
[22]  Z[STK](N)<--MKREF(S)
      $
```

```
        $NODE(A,B,C,D)R;;
   [1]  R<--MKSTR(FATHER:A,RIGHTBROTHER:B,FIRSTSON:C,VALUE:D)
        $


        $F(X)R;;
   [1]  EQUAL(X,"S",R<--0)→7
   [2]  EQUAL(X,"E",R<--0)→7
   [3]  EQUAL(X,"T",R<--1)→7
   [4]  EQUAL(X,"A",R<--1)→7
   [5]  EQUAL(X,"+",R<--0)→7
   [6]  EQUAL(X,"RPAD",R<--0)→7
        $


        $G(X)R;;
   [1]  EQUAL(X,"S",R<--0)→7
   [2]  EQUAL(X,"E",R<--0)→7
   [3]  EQUAL(X,"T",R<--1)→7
   [4]  EQUAL(X,"A",R<--2)→7
   [5]  EQUAL(X,"+",R<--0)→7
   [6]  EQUAL(X,"RPAD",R<--0)→7
        $

        RULE1<--MKSTR(LP:"E",RP:MKROW("E","+","T"))
        RULE2<--MKSTR(LP:"T",RP:MKROW("A","T"))
        RULE3<--MKSTR(LP:"E",RP:MKROW("T"))
        RULE4<--MKSTR(LP:"T",RP:MKROW("A"))
        GRAMMAR<--MKREF(MKROW(RULE1,RULE2,RULE3,RULE4))
        X<--MKREF(MKROW("A","A","+","A","+","A","RPAD"))
        Y<--PARSE(X,GRAMMAR,F,G)
        SHOWPREFIX(Y)
S(E(E(E(T(A,T(A))),+,T(A)),+,T(A)),RPAD)
        SHOWSUFFIX(Y)
(((((A,(A)T)T)E,+,(A)T)E,+,(A)T)E,RPAD)S
        SHOWCONSTANTDEPTH(Y)
( S)( E RPAD)( E + T)( E + T A)( T A)( A T)( A)
```

G Complex and Rational Arithmetic

In this extension we redefine the arithmetic operators to accept arguments that are a pair of complex numbers (complex = <u>struct</u>(rp:<u>real</u> , ip:<u>real</u>, type:"comp")) or a pair of rationals (rational= <u>struct</u>(num:<u>integer</u>, den:<u>integer</u>,type:"ratio")) as well as the atomic arguments they previously accepted. Complex is the constructor for complex numbers and SDIV constructs rationals when called with a pair of integer operands. Hence the infix operator "/" acts sometimes as a constructor and sometimes as a normal divide operator. We note that the rational constructor function always produces a rational whose num and den components are coprime, using the gcd (greatest common divisor) function to this end. The functions and some output produced by them are as follows:

```
     DIV<--SDIV
     MUL<--SMUL
     ADD<--SADD
     SUB<--SSUB

     $COMPLEX(A,B)R;;
[1]  R<--MKSTR(RP:A,IP:B,TYPE:"COMP")
     $


     $RATIO(A,B)R;G;
[1]  G<--GCD(A,B)
[2]  R<--MKSTR(NUM:IDIV(A,G),DEN:IDIV(B,G),TYPE:"RATIO")
     $
```

```
      $SDIV(A,B)R;D;
[1]   ((ILK(A)=2)×(ILK(B)=2))→7
[2]   (ILK(A)=1)→5
[3]   (A[TYPE]="COMP")→9
[4]   (A[TYPE]="RATIO")→12
[5]   R<--DIV(A,B)
[6]   →13
[7]   R<--RATIO(A,B)
[8]   →13
[9]   D<--(B[RP]×B[RP])+B[IP]×B[IP]
[10]  R<--COMPLEX(((X[IP]×Y[IP])+X[RP]×Y[RP])/D,((X[IP]×Y[RP])-X[RP]×Y[IP])/D)
[11]  →13
[12]  R<--(A[NUM]×B[DEN])/A[DEN]×B[NUM]
      $


      $SMUL(A,B)R;;
[1]   ((ILK(A)=1)+ILK(A)=2)→4
[2]   (A[TYPE]="COMP")→8
[3]   (A[TYPE]="RATIO")→6
[4]   R<--MUL(A,B)
[5]   →9
[6]   R<--(A[NUM]×B[NUM])/A[DEN]×B[DEN]
[7]   →9
[8]   R<--COMPLEX((A[RP]×B[RP])-A[IP]×B[IP],(A[RP]×B[IP])+A[IP]×B[RP])
      $


      $SADD(A,B)R;CRPR,G;
[1]   IPLUS(ILK(A)=1,ILK(A)=2)→4
[2]   (A[TYPE]="COMP")→9
[3]   (A[TYPE]="RATIO")→6
[4]   R<--ADD(A,B)
[5]   →10
[6]   CRPR<--(A[NUM]×B[DEN])+A[DEN]×B[NUM]
[7]   R<--CRPR/A[DEN]×B[DEN]
[8]   →10
[9]   R<--COMPLEX(A[RP]+B[RP],A[IP]+B[IP])
      $
      $GCD(X,Y)R;;
 [1]  X→3
 [2]  X<--0-X
 [3]  Y→5
 [4]  Y<--0-Y
 [5]  EQUAL(Y,1,R<--1)→14
 [6]  EQUAL(X,1,R<--1)→14
 [7]  EQUAL(X,0,R<--Y)→14
 [8]  EQUAL(Y,0,R<--X)→14
 [5]  (X-Y)→12
 [6]  Y<--Y-X
 [7]  →5
 [8]  X<--X-Y
 [9]  →5
      $
```

```
      $PRINT(X);;
 [1]  ((ILK(X)=2)+(ILK(X)=3))→12
 [2]  TYPE(X(1))
 [3]  (X[TYPE]="RATIO")→9
 [4]  IFLJM(6,X[IP])
 [5]  TYPE("+")
 [6]  TYPE(X[IP])
 [7]  TYPE("I
")
 [8]  →13
 [9]  TYPE("/")
 [10] X[DEN]
 [11] →13
 [12] X
      $
```

```
        A<--COMPLEX(1.0,-2.5)
        B<--COMPLEX(3.0,4.0)
        PRINT(A)
1.0-2.5I
        PRINT(B)
3.0+4.0I
        PRINT(A+B)
4.0+1.5I
        PRINT(A×B)
1.3E1-3.5I
        PRINT(A/B)
-2.8E-1-4.6E-1I
        A<--4/7
        B<--3/2
        PRINT(A/B)
8/21
        PRINT(A×B)
6/7
        PRINT(A+B)
29/14
        C<--24/7
        PRINT(A+C)
4/1
```

H. Block Structure and Own Variables

The next extension we describe is not implemented in the current version of CEL (though the additions to the current CEL that are necessary to make it implementable are straightforward). The base language of CEL has only two levels of block structure - variables are either global or local to a function. The following extension would add multilevel block structure and scoping of variables (in the ALGOL sense).

The primary data structure that we use is an environment, defined as follows:

Env = *struct*(father:block, current:*struct*)

Block = NIL *or* env.

We note that the type of env[current] is not completely specified. This is convenient in view of the way data of type env are to be constructed and is made possible by the absence of declarations in CEL. We first define functions that open and close blocks and decide which variable an identifier refers to in a particular environment.

```
     $BEGIN(L);I,X;
[1]  I<--LNGTH(L)
[2]  X<--MKNST(I)
[3]  J<--1
[4]  (J-I)→8
[5]  NAME(X(J),L(J))
[6]  J<--J+1
[7]  →4
[8]  CURRENT<--MKREF(MKSTR(FATHER:CURRENT,CURRENT:X))
     $

     $END();;
[1]  CURRENT<--VLPTR(CURRENT)[FATHER]
     $
```

```
    $SELECT(A)R;L;
[1] L<--CURRENT
[2] ELEMENT(A,VLPTR(L)[CURRENT])→7
[3] (VLPTR(L)[FATHER]=NIL)→6
[4] L<--VLPTR(L)[FATHER]
[5] →2
[6] ERROR()
[7] R<--VLPTR(L)[CURRENT;A]
    $
```

The argument to begin is a row of the identifiers to be local to the block being opened. The unimplemented (as yet) library function MKNST creates a struct, x, whose length is the number of local identifiers. The library function NAME then attaches the names of the elements of l to the components of x. Finally, the environment thus created is made the current environment with, as father, the previous environment. End simply transforms the current environment to the previous one. Select, given an identifier, finds the version of it whose scope includes the current block using the (currently unimplemented) library function ELEMENT which determines whether a struct has a component with a specified name. One presumes that this extension would be used in conjunction with a syntax mapper in which <identifier> was converted to the equivalent in the base syntax of select (<identifier>). We note that we can easily adjust this extension, by making 'current' an argument to begin, end and select, so that we can deal with multiple parallel environments.

We can use a similar technique to obtain the effect of own variables, i. e. , variables local to a function whose lifetime properly contains the time during which the function is being executed. To do this we define a data type

funcwithowns = struct(func:function, vars:struct).

Now we must define a function that is to take own variables with a header such as "$f(x) . . . " where x is a struct whose component names are those of the formal parameters and own variables of f. Within f, we use a mechanism like the function select (but simpler, since environments are not nested) to get at the parameters and own variables of f. To call f, we write "call(fprime, x_1, . . . , x_n)" where the x_i are the arguments to f, fprime = struct(func:f, owns:b), and the syntax mapper transforms "call(fprime, x_1, . . . , x_n)" to f(fprime[vars]) and sets the components of fprime[vars] corresponding to arguments appropriately.

SECTION VII

The Implementation of CEL

CEL is currently implemented in approximately two thousand lines of code on a Digital Equipment Corporation PDP-10 Computer, a one-address machine with sixteen accumulators and a sizeable instruction set. The implementation provides the facilities described above via several data structures. Of particular interest are a pair of push down stacks which are used to drive program execution and a large data area which contains all linked structures used - including all program variables as well as program text. This space is garbage collected by means of a modification of the algorithm described by Schorr and Waite in [25] (see also [17], p. 417).

We hope, as noted previously, to make several eventual improvements in this implementation. In particular, we plan to include a Brooker and Morris-like syntax definition mechanism and text editing and debugging facilities like those of APL. Some less important changes are also intended, including the addition of a number of new library functions and the improvement of the storage management algorithms currently in use.

SECTION VIII

Conclusions

We hope that it is clear from the examples given above that extensible languages in general, and CEL in particular, provide mechanisms that make it possible to deal conveniently with many diverse problem areas. We note that each of the examples of Section VI was coded and debugged in less than a day's time (often a good deal less). By contrast, the implementers of FORMULA ALGOL required eight man years to produce a system containing these facilities [32]. We do not have enough experience with programming in extensible languages to provide a basis for comparing their practical utility with that of the shell and special purpose languages, particularly in large applications. But we can point to the relative ease of implementing extensible languages and programming in them (especially when conversational features are included) as very significant advantages of extensible languages. Even where object program efficiency is an important consideration, it may often turn out that a compilation facility (together with optional declarations in a typeless language) will make object program efficiency quite adequate.

Appendix A

Library Functions of CEL

The following library functions are included in CEL's current implementation:

1. ASGNC (respectively ASIGN) takes two arguments and sets the value of the first as the second (respectively a copy of the second). ASIGN is invoked by the infix operator <--.

2. BRNCH, if called with two arguments x and y is equivalent to IFGJM(y, x); if called with one argument x it is equivalent to GOTO. It is invoked by the infix operator-->.

3. EQUAL(x, y) = if x is the same as y in value and type then 1 and otherwise 0. It is invoked by the infix operator =

4. GOTO takes a single integer argument and sets the program counter to the value of this argument.

5. IDIV (respectively IMULT, IPLUS, ISUB) takes two integer arguments x and y and returns $[x \div y]$ (respectively the product, sum, difference of x and y).

6. IFGJM (respectively IFLJM, IFZJM) takes two integer arguments and sets the value of the program counter as the first if the second is positive (respectively negative, zero).

7. ILK(x) returns an integer code for the type of x as follows:

NIL	0
real	1
integer	2
literal	3
struct	4
ref	5

row	6
oref	7
identifier	8
delimiter	9
code string pair	10
defined function	11
library function	12
packed identifiers	13
statement	14
locked function	15

8. LNGTH takes a single row or struct argument and returns its length.

9. MKNRW(n), n a positive integer, returns a row x of length n satisfying $(\forall 1 \leq i \leq n)$ $(x(i) = \mathrm{NIL})$

10. MKREF(x) returns a pointer to x.

11. MKROW takes an arbitrary positive number of arguments and returns a row (whose length is the number of arguments) of copies of their values.

12. MKSTR takes a set of arguments $\{L_i:V_i\}_{i=1}^{n}$, $n \geq 0$, and returns a struct x of length n whose ith component is named L_i and has as value a copy of V_i.

13. NOTEQ(x, y) = if x is the same as y in value and type then 0 and otherwise 1. It is invoked by the infix operator $\neq$. .

14. RDIV (respectively RMULT, RPLUS, RSUB) takes two real arguments and returns their quotient (respectively product, sum, difference).

15. SDIV (respectively SMUL, SPLUS, SSUB) takes two real or integer arguments, converts the second to the type of the first and invokes IDIV or RDIV (respectively IMUL or RMUL, IPLUS or RPLUS, ISUB or SSUB) as appropriate to do the calculation. It is invoked

by the infix binary operator / (respectively *, +, -).

16. TYPE takes a single argument and outputs it to the user's teletype if it is <u>real</u>, <u>integer</u> or <u>literal</u>.

17. VLPTR(x) returns, for x a <u>ref</u>, the datum at which x points.

Appendix B

F and G Functions and Precedence Matrix for a Front End Syntax of CEL

G: S ::= E | I:E

E ::= T α E | α E | T

T ::= I | C | (S) | T() | T(S, . . . , S) | T[I; . . . ; I]

g		1	3	3	1	3	3	3	1	3	3
f		:	α	(	)	;	[	]	,	C	I
2	:		<	<	>		<		>	<	<
2	α		<	<	>		<		>	<	<
1	(		<	<	=		<		=	<	<
4	)		>	>	>		>		>		
3	;										=
3	[										=
4	]		>	>	>		>		>		
1	,		<	<					=	<	<
4	C		>	>	>		>		>		
4	I	=	>	>	>	=	>	=	>		

REFERENCES

1. Berry, Paul, APL/360 Primer, International Business Machines Corporation, (1969).

2. Bobrow, Daniel G. (ed.), Symbol Manipulation Languages and Techniques, North Holland Publishing Company, Amsterdam, (1968).

3. Brooker, R.A., Morris, D., "A General Translator Program for Phrase Structure Languages," Journal of the ACM, vol. 9, no. 1, (Jan. 1962), p. 1 ff.

4. Cheatham, T.E. Jr., "The Introduction of Definitional Facilities into Higher Level Languages," Proc. AFIPS FJCC, vol. 29, (Nov. 1966).

5. ________________, Lectures and Notes for Applied Mathematics 223, Harvard University, (Spring 1969).

6. ________________, et al., "On the Basis for ELF: An Extensible Language Facility," Proc. AFIPS FJCC, vol. 33, Part II, (Dec. 1968).

7. ________________, The Theory and Construction of Compilers, Second Edition, Computer Associates (1967 and Subsequent Revisions).

8. Christensen, Carlos, et al., (ed.), SIGPLAN Notices: Proceedings of the Extensible Languages Symposium, vol. 4, (Aug. 1969).

9. Conway, M.E., "Design of a Separable Transition Diagram Compiler," Communications of the ACM, vol. 6, no. 4, (July 1963), p. 396 ff.

10. Feldman, J.A. and Gries, D., "Translator Writing Systems," Communications of the ACM, vol. 11, no. 2, (Feb. 1968).

11. Floyd, Robert W., "Syntactic Analysis and Operator Preced-

ence," Journal of the ACM, vol. 10, no. 3, (1966), pp. 316-333.

12. Galler, B.A. and Perlis, A.J., "A Proposal for Definitions in ALGOL," Communications of the ACM, vol. 10, no. 4, (April 1967).

13. Garwick, Jan V., "GPL: A Truly General Purpose Language," Communications of the ACM, vol. 11, no. 9, (Sept. 1968).

14. Gold, Michael M., "Time Sharing and Batch - An Experimental Comparison of Their Values in a Problem-solving Situation," Communications of the ACM, vol. 12, no. 5, (May 1969).

15. Iverson, Kenneth E., A Programming Language, John Wiley and Sons, Inc., New York, (1962).

16. Jorrand, Phillipe, "BASEL, the Base Language for an Extensible Language Facility," SIGPLAN Extensible Languages Symposium, War Memorial Auditorium, Boston, (May 13, 1969).

17. Knuth, Donald E., The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, (1968).

18. ________________, The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Addison-Wesley, (1969).

19. Leavenworth, B.M., "Syntax Macros and Extended Translation," Communications of the ACM, vol. 9, no. 11, (Nov. 1966).

20. Landin, P.J., "The Mechanical Evaluation of Expressions," Computer Journal, vol. 6, no. 4, (Jan. 1964), p. 308 ff.

21. McCarthy, John, "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I,"

Communications of the ACM, vol. 3, no. 3, (March 1960), p. 184 ff.

22. Mealy, George H., Personal Communication, (March 1970).

23. Reynolds, Jon C., "Gedanken - A Simple Typeless Language Which Permits Functional Data and Co-routines," Argonne National Laboratories, Argonne, Illinois, (May 1969).

24. Sackman, H., Erikson, W. J. and Grant, E. E., "Exploratory Experimental Studies Comparing On Line and Off Line Programming Performance," Communications of the ACM, vol. 12, no. 1, (Jan. 1968).

25. Schorr, H. and Waite, W. M., "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures," Communications of the ACM, vol. 10, no. 8, (Aug. 1967).

26. Standish, Thomas A., "A Data Definition Facility for Programming Languages," Carnegie Institute of Technology, Pittsburgh, (May 1967), (Doctoral Dissertation).

27. ____________________, "An Essay on APL," Carnegie Institute of Technology, (March 1969).

28. ____________________, Lectures and Notes for Applied Mathematics 260-261, Harvard University, (Fall-Spring, 1969-70).

29. ____________________, "A Preliminary Sketch of a Polymorphic Programming Language," Centro De Calculo Electronico, Universidad Nacional Autonoma De Mexico, Mexico City, (July 1968).

30. ____________________, "Some Compiler-Compiler Techniques for Use in Extensible Languages," SIGPLAN Extensible Languages Symposium, War Memorial Auditor-

ium, Boston, (May 13, 1969).

31. ____________________, "Some Features of PPL - A Polymorphic Programming Language," SIGPLAN Extensible Languages Symposium, War Memorial Auditorium, Boston, (May 13, 1969).

32. ____________________, Personal Communication, (April 22, 1970).

33. Thompson, F. B., "REL: A Rapidly Extensible Language," California Institute of Technology, Pasadena, California, (Feb. 1969).

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Harvard University Cambridge, Massachusetts	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP N/A

3. REPORT TITLE

THE DESIGN AND IMPLEMENTATION OF A CONVERSATIONAL EXTENSIBLE LANGUAGE

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
None

5. AUTHOR(S) *(First name, middle initial, last name)*

Jay M. Spitzen

6. REPORT DATE May 1970	7a. TOTAL NO. OF PAGES 77	7b. NO. OF REFS 70
8a. CONTRACT OR GRANT NO. F19628-68-C-0101	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-70-141	
b. PROJECT NO.		
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	
d.		

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sales; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Directorate of Systems Design and Development, Hq Electronic Systems Division (AFSC), L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT

This report describes CEL, a conversational extensible language. Its syntax, data, control structures and conversational features are presented and compared to those of other languages. Its use is illustrated by means of several examples in the areas of list processing, polynomial arithmetic, formula manipulation, vector arithmetic, trees and syntax analysis, complex and rational arithmetic and block structure and own variables.

DD FORM 1 NOV 65 1473

Unclassified
Security Classification

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Extensible Language Conversational Language Data Definition Facility						

Unclassified
Security Classification